



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н. Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н. Э. Баумана)

---

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии» (ИУ7)

## Отчет по лабораторной работе №1 по курсу «Анализ алгоритмов»

*«Расстояние Левенштейна и Дamerau-Левенштейна»*

Группа: ИУ7-53Б

Студент:

\_\_\_\_\_  
(Подпись, дата) Дьяченко А. А.  
(Фамилия И. О.)

Преподаватель:

\_\_\_\_\_  
(Подпись, дата) Строганов Д. В.  
(Фамилия И. О.)

Москва, 2023 г.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ . . . . .	4
1 Аналитическая часть . . . . .	5
1.1 Нерекursивный алгоритм поиска расстояния Левенштейна . . . .	5
1.2 Нерекursивный алгоритм поиска расстояния Дамерау-Левенштейна . . . . .	6
1.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна . . . . .	8
1.4 Рекурсивный алгоритм с кэшированием для нахождения расстояния Дамерау-Левенштейна . . . . .	9
2 Конструкторская часть . . . . .	10
2.1 Разработка нерекursивного алгоритма нахождения расстояния Левенштейна . . . . .	10
2.2 Разработка нерекursивного алгоритма нахождения расстояния Дамерау-Левенштейна . . . . .	11
2.3 Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна . . . . .	12
2.4 Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем . . . . .	13
3 Технологическая часть . . . . .	14
3.1 Требования к ПО . . . . .	14
3.2 Средства реализации . . . . .	14
3.3 Сведения о модулях программы . . . . .	14
3.4 Реализация алгоритмов . . . . .	15
4 Исследовательская часть . . . . .	19
4.1 Технические характеристики . . . . .	19
4.2 Пример работы программы . . . . .	19
4.3 Время выполнения реализованных алгоритмов . . . . .	20
4.4 Занимаемая память реализованных алгоритмов . . . . .	21
4.5 Сравнительный анализ алгоритмов . . . . .	21
4.6 Вывод . . . . .	22

## ВВЕДЕНИЕ

Целью данной лабораторной работы является изучение расстояния Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна (редакционное расстояние) и его модификация - расстояние Дамерау-Левенштейна, представляют собой метрики, используемые для измерения различий между двумя последовательностями символов. Они определяют минимальное количество односимвольных операций (вставка, удаление, замена и транспозиция), необходимых для преобразования одной последовательности символов в другую. Эти метрики были разработаны советским математиком Владимиром Левенштейном в 1965 году и модифицированы впоследствии с учетом операции транспозиции символов, получив название расстояния Дамерау-Левенштейна.

Расстояние Левенштейна и его модификация имеют широкий спектр применений в различных областях, включая компьютерную лингвистику (автозамена, исправление ошибок), биоинформатику (анализ генома, белковых последовательностей).

Задачи лабораторной работы:

- изучение алгоритмов для вычисления расстояния Левенштейна и Дамерау-Левенштейна;
- реализация с использованием динамического программирования;
- сравнительный анализ требуемого времени в тактах процессора и занимаемой памяти;
- подготовка отчёта по лабораторной работе.

## 1 Аналитическая часть

Расстояние Дameraу-Левенштейна представляет собой метрику, измеряющую разницу между двумя строками, путем подсчета минимального числа операций, необходимых для преобразования одной строки в другую. Эти операции включают в себя вставку, удаление, замену символов и дополнительную относительно расстояния Левенштейна операцию, которая называется транспозицией, которая представляет собой перестановку двух соседних символов в строке.

Обозначения редакторских операций:

- а) I - вставка символа (insert)
- б) D - удаление (delete)
- в) R - замена (replace)
- г) M - совпадение двух символов (match)
- д) T - транспозиция двух соседних символов (transposition)

I, D, R, T - штраф 1 (в преобразуемой строке).

Совпадение символа с самим собой не имеет дополнительной стоимости и оценивается как 0.

Эти операции и их стоимости позволяют вычислить расстояние Дameraу-Левенштейна между двумя строками, что является важным инструментом в областях, где требуется измерять сходство или различие между текстовыми последовательностями.

### 1.1 Нерекурсивный алгоритм поиска расстояния Левенштейна

Пусть  $L1$  - длина строки  $S_1$ ,  $L2$  - длина строки  $S_2$ .  $S_1[1..i]$  - подстрока  $S_1$  с длиной  $i$  символов, начиная с первого,  $S_2[1..j]$  - подстрока  $S_2$  длиной  $j$  символов, начиная с первого.

Расстояние Левенштейна может быть найдено с помощью формулы:

$$D(i, j) = \begin{cases} \max(i, j) & \text{если } \min(i, j) = 0, \\ \min\{ \\ \quad D(i, j - 1) + 1, \\ \quad D(i - 1, j) + 1, \\ \quad D(i - 1, j - 1) + m(S_1[i], S_2[j]), & \text{иначе} \\ \} \end{cases}, \quad (1)$$

где  $m(S_1[i], S_2[i])$  равна нулю, если  $S_1[i] = S_2[i]$  и единице в противном случае.

Для оптимизации вычисления расстояния между строками используется матрица промежуточных значений. Её размерность определяется как  $(len(S_1) + 1)(len(S_2) + 1)$ . Каждая ячейка матрицы, обозначенная как  $matrix[j, j]$ , содержит значение расстояния между подстроками  $S_1[1...i]S_2[1...j]$ . Первая строка и первый столбец этой матрицы представляют собой тривиальные случаи и соответствуют наибольшему  $i$  или  $j$  в соответствующей строке или столбце.

Матрица заполняется в соответствии со следующей формулой:

$$M[i][j] = \min \begin{cases} M[i - 1][j] + 1, \\ M[i][j - 1] + 1, \\ M[i - 1][j - 1] + m(S_1[i], S_2[j]) \end{cases}. \quad (2)$$

Расстоянием Левенштейна будет значение в самой правой нижней ячейке матрицы с индексами  $i = len(S_1), j = len(S_2)$ .

## 1.2 Нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между двумя строками, обозначенными как  $S_1$  и  $S_2$ , может быть вычислено с использованием формулы 3. Эта формула имеет следующий вид:

$$D(i, j) = \begin{cases} \min(i, j) = 0, & \text{если } \max(i, j) = 0, \\ \min \begin{cases} D(i, j - 1) + 1, \\ D(i - 1, j) + 1, \\ D(i - 1, j - 1) + m(S_1[i], S_2[j]), \end{cases} & \text{иначе} \\ \begin{cases} D(i - 2, j - 2) + 1, & \text{если } i, j > 1; \\ & S_1[i] = S_2[j - 1]; \\ & S_1[i - 1] = S_2[j] \end{cases} & \\ \infty, & \text{иначе} \end{cases}, \quad (3)$$

Эта формула представляет собой рекуррентное соотношение, которое позволяет вычислить расстояние между двумя строками, учитывая вставки, удаления, замены и транспозиции символов. В результате выполнения этой формулы получается матрица, где значение в ячейке с индексами  $i = \text{len}(S_1)$  и  $j = \text{len}(S_2)$  представляет собой расстояние Дамерау-Левенштейна между исходными строками  $S_1$  и  $S_2$ .

Формула 3 выводится на основе аналогичных рассуждений, что и формула 1, но с учетом дополнительной редакторской операции - транспозиции символов. Эта формула позволяет эффективно находить расстояние Дамерау-Левенштейна и широко используется в задачах редактирования и сравнения строк.

### 1.3 Рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна

Рекурсивный алгоритм для вычисления расстояния Дameraу-Левенштейна реализует формулу 3. Функция  $D$  определяется следующим образом:

- 1) Перевод из пустой строки в пустую строку не требует никаких операций и, следовательно, имеет стоимость равную нулю.
- 2) Перевод из пустой строки в строку  $S_1$  требует выполнения  $|S_1|$  операций, каждая из которых добавляет один символ к строке  $S_1$ . Таким образом, общая стоимость такого перевода равна  $|S_1|$ .
- 3) Перевод из строки  $S_1$  в пустую строку аналогично требует выполнения  $|S_1|$  операций удаления, каждая из которых удаляет один символ из строки  $S_1$ . Следовательно, общая стоимость такого перевода также равна  $|S_1|$ .
- 4) Для перевода из строки  $S_1$  в строку  $S_2$  требуется выполнить последовательность операций (удаление, вставка, замена, транспозиция) в определенной последовательности.

Полагая  $S'_1$  и  $S'_2$  как строки  $S_1$  и  $S_2$  без их последних символов соответственно, цена преобразования из строки  $S_1$  в строку  $S_2$  может быть выражена следующими случаями:

- а) Сумма цены преобразования строки  $S'_1$  в  $S_2$  и цены операции удаления, необходимой для преобразования  $S'_1$  в  $S_1$ .
- б) Сумма цены преобразования строки  $S_1$  в  $S'_2$  и цены операции вставки, необходимой для преобразования  $S'_2$  в  $S_2$ .
- в) Сумма цены преобразования из  $S'_1$  в  $S'_2$  и цены операции замены, предполагая, что последние символы  $S_1$  и  $S_2$  разные.
- г) Цена преобразования из  $S'_1$  в  $S'_2$ , предполагая, что последние символы  $S_1$  и  $S_2$  совпадают.
- д) Сумма цены преобразования  $S''_1$  в  $S''_2$ , предполагая, что последние два символа  $S_1$  можно преобразовать путем транспозиции в два последних символа  $S_2$ .

Таким образом, рекурсивный алгоритм разбивает задачу на более мелкие подзадачи и находит оптимальное решение, учитывая все возможные варианты операций.

## 1.4 Рекурсивный алгоритм с кэшированием для нахождения расстояния Дameraу-Левенштейна

Рекурсивный алгоритм с кэшированием является улучшенной версией рекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна. Он использует ту же формулу 3, но с добавлением механизма кэширования, чтобы избежать повторных вычислений и снизить вычислительную сложность.

Функция  $D$  в этой версии алгоритма хранит результаты вычислений в специальной матрице кэша. Каждый элемент этой матрицы соответствует значениям  $D(i, j)$  для определенной пары индексов  $i$  и  $j$ . Если значение уже было вычислено, оно сохраняется в кэше, и при необходимости оно просто извлекается из кэша, вместо того чтобы пересчитываться заново.

Такой подход существенно ускоряет процесс вычисления расстояния Дameraу-Левенштейна, особенно при работе с большими строками. Эффективное использование кэширования позволяет избежать множественных повторных вычислений и сделать алгоритм более производительным.

Таким образом, рекурсивный алгоритм с кэшированием представляет собой оптимизированную версию рекурсивного алгоритма, способную эффективно находить расстояние Дameraу-Левенштейна между двумя строками.



## 2 Конструкторская часть

### 2.1 Разработка нерекурсивного алгоритма нахождения расстояния Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Левенштейна представлена на рисунке 1.

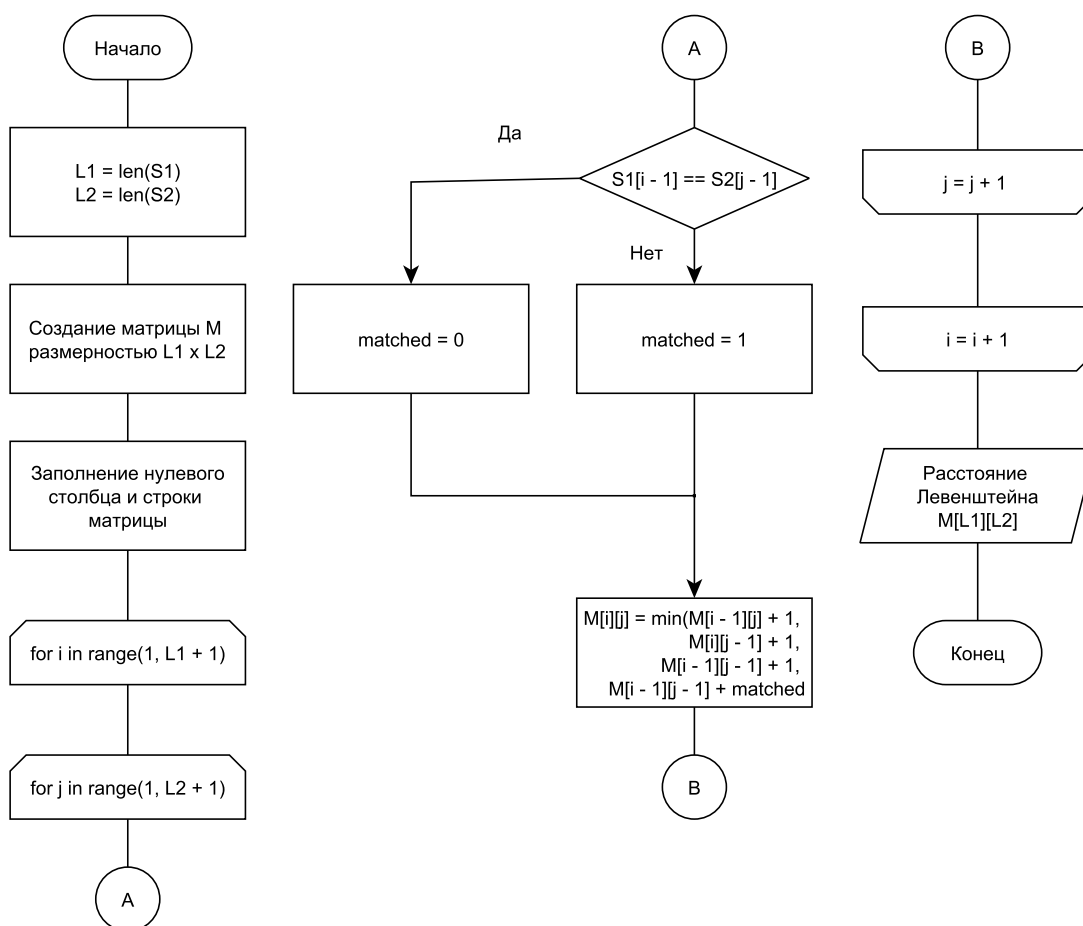


Рисунок 1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

## 2.2 Разработка нерекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна представлена на рисунке 2.

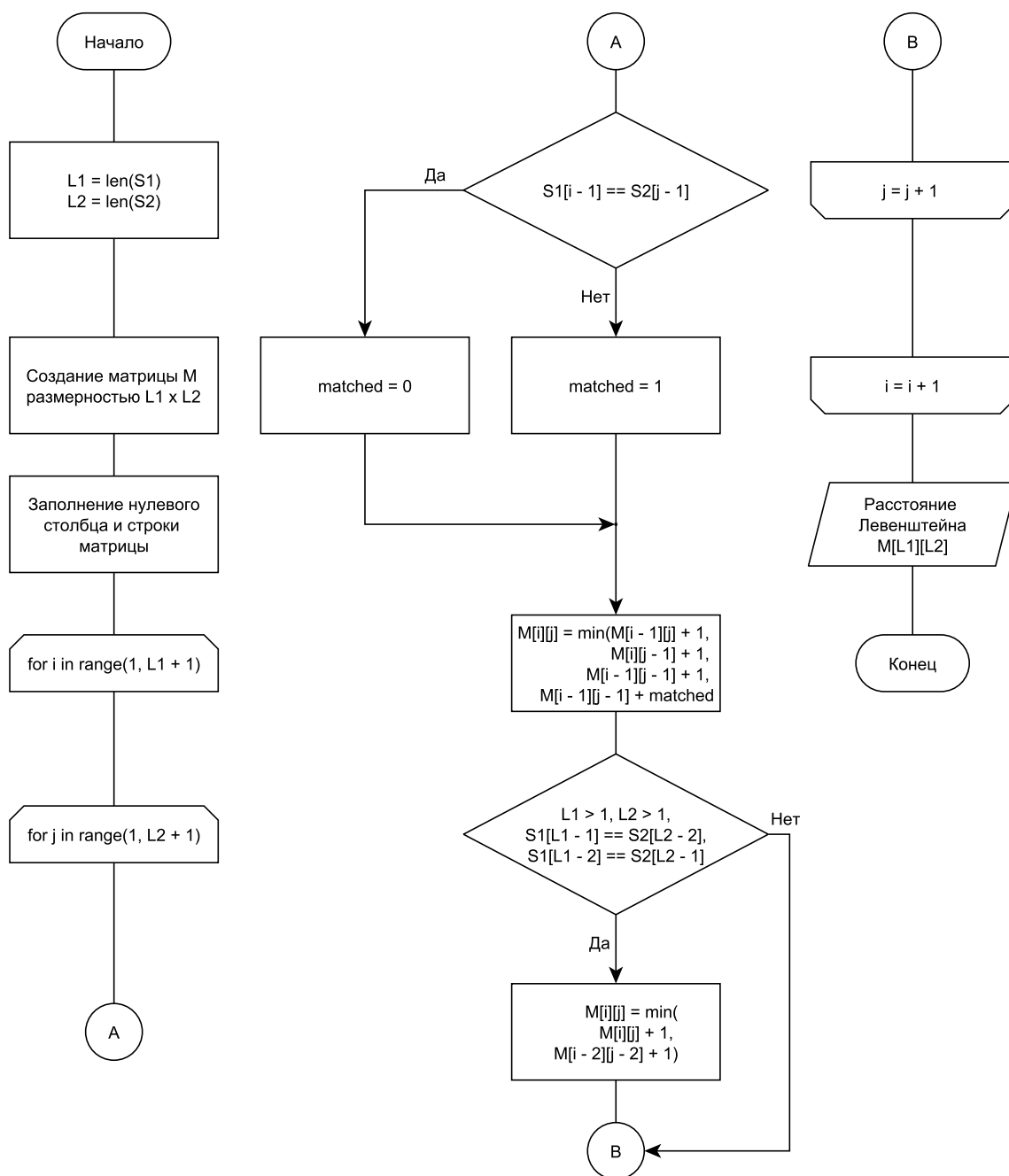


Рисунок 2 – Схема нерекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна

## 2.3 Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна представлена на рисунке 3.

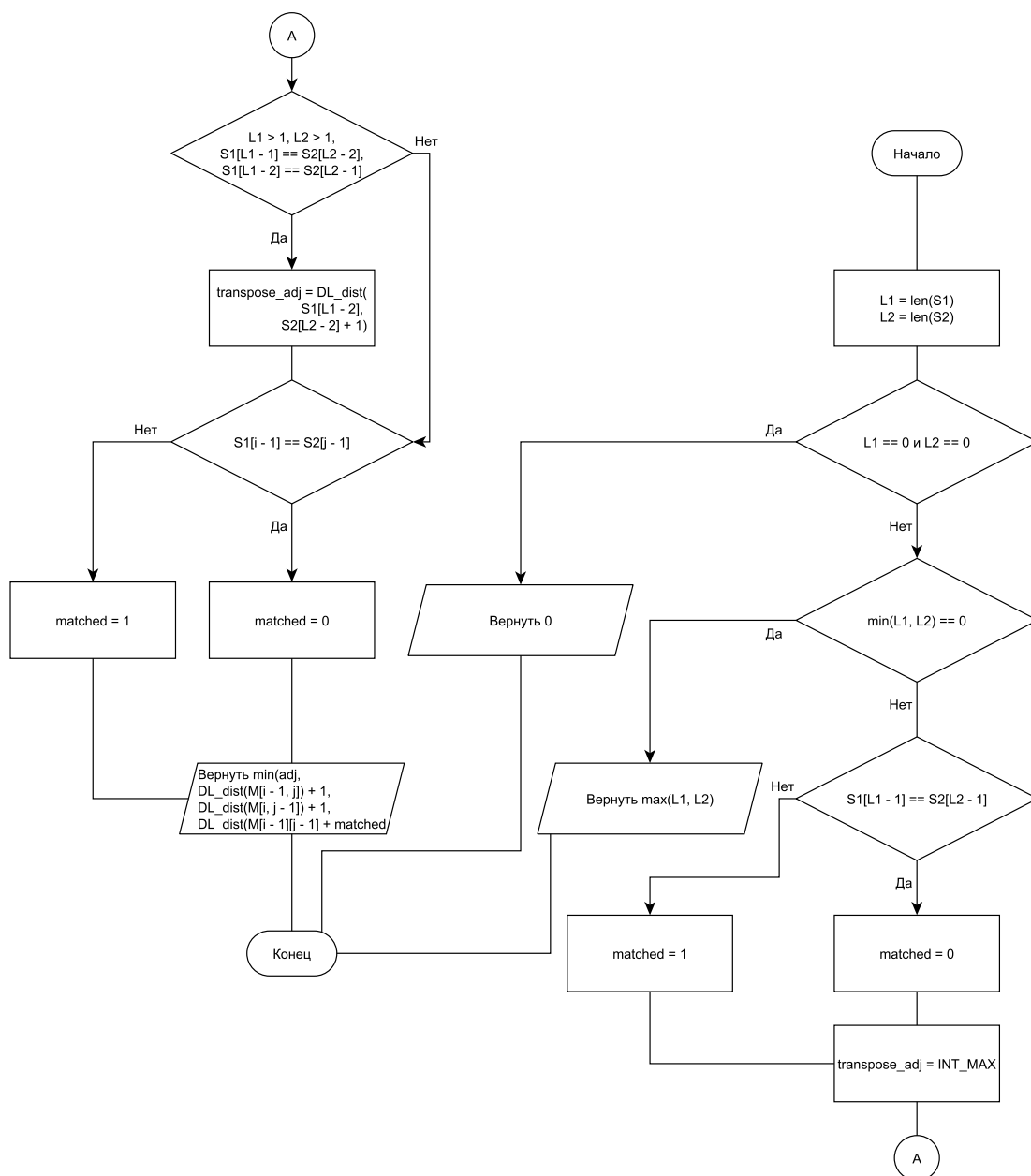


Рисунок 3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

## 2.4 Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем

Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем представлена на рисунке 4.

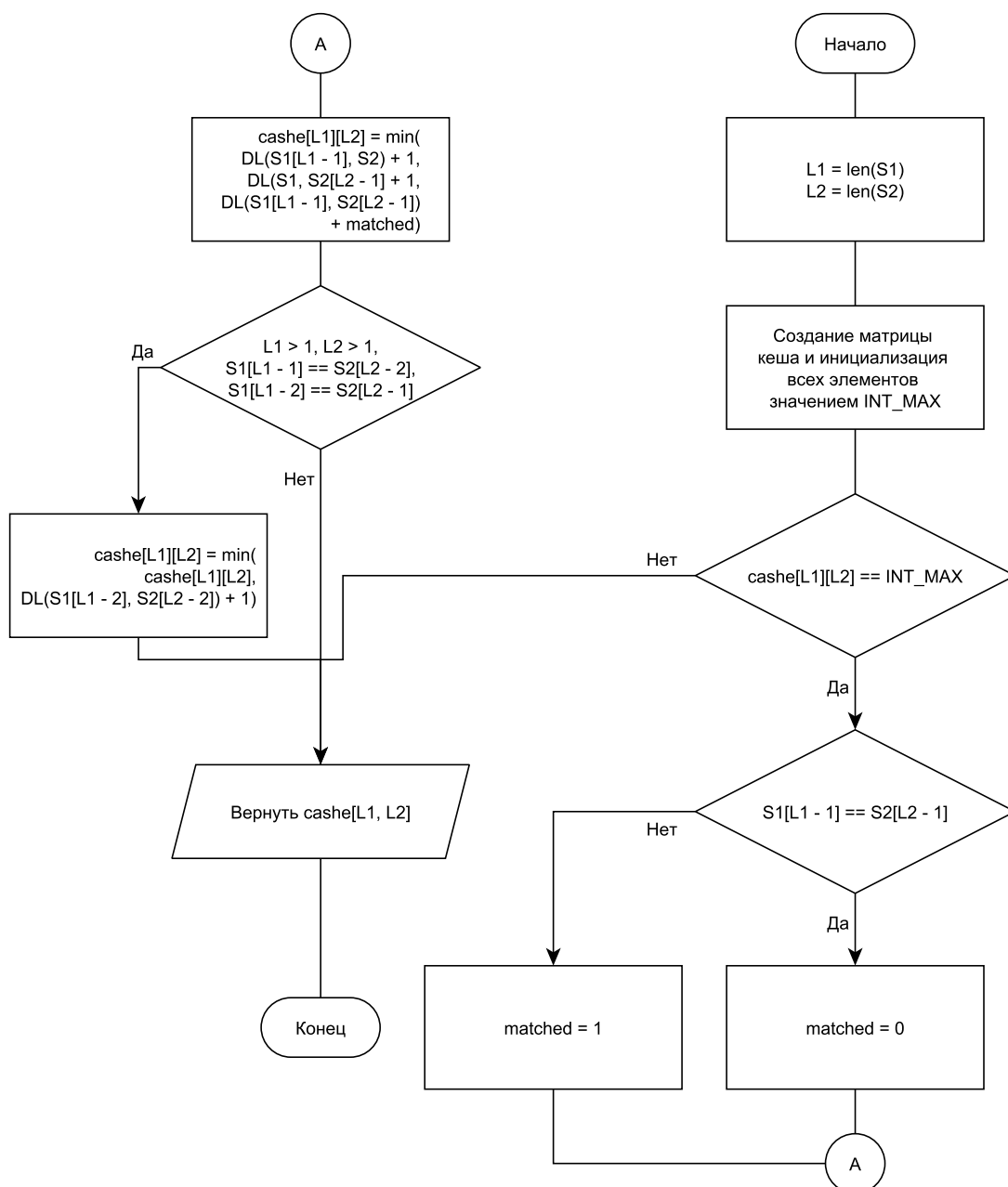


Рисунок 4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем

## 3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

### 3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки на русском или английском языке в любом регистре;
- на выходе – искомое расстояние для трёх методов и их время выполнения.

### 3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран C++.

Данный выбор обусловлен поддержкой языком парадигмы объектно – ориентированного программирования и наличием библиотек для точного замера времени.

Время выполнения реализации алгоритмов было измерено с помощью библиотеки *chrono*.

### 3.3 Сведения о модулях программы

Программа состоит из трех программных модулей:

- 1) `main.cpp` – главный модуль программы, содержащий функцию `main` и замер времени;
- 2) `algorithm.cpp`, `algorithm.h` – модуль с реализацией алгоритмов;
- 3) `menu.cpp`, `menu.h` – модуль с выбором режима работы программы;
- 4) `tests.cpp`, `tests.h` – модуль с тестированием программы.

### 3.4 Реализация алгоритмов

В листинге 1 приведена реализация итеративного алгоритма нахождения расстояния Левенштейна.

```
1  int DistanceSolver::iterativeLevenshtein() {
2      int L1 = S1.length();
3      int L2 = S2.length();
4
5      // Create and initialize a matrix of size (L1+1) x (L2+1)
6      vector<vector<int>> dp(L1 + 1, vector<int>(L2 + 1, 0));
7
8      // Initialize the first row and the first column of the matrix
9      for (int i = 0; i <= L1; i++) {
10         dp[i][0] = i;
11     }
12     for (int j = 0; j <= L2; j++) {
13         dp[0][j] = j;
14     }
15
16     // Fill in the matrix
17     for (int i = 1; i <= L1; i++) {
18         for (int j = 1; j <= L2; j++) {
19             int cost = (S1[i - 1] == S2[j - 1]) ? 0 : 1;
20             dp[i][j] = min(min(dp[i - 1][j] + 1, dp[i][j - 1] + 1),
21                             dp[i - 1][j - 1] + cost);
22         }
23     }
24
25     // The Levenshtein distance is found in the bottom-right corner
26     // of the matrix
27     return dp[L1][L2];
28 }
```

Листинг 1 – Итеративный алгоритм нахождения расстояния Левенштейна

В листинге 2 приведена реализация итеративного алгоритма нахождения расстояния Дамерау-Левенштейна.

```
1  int DistanceSolver::iterativeDamerauLevenshtein() {
2      int L1 = S1.length();
3      int L2 = S2.length();
4
5      // Create a 2D vector for dynamic programming
6      vector<vector<int>> dp(L1 + 1, vector<int>(L2 + 1, 0));
7
8      // Initialize the first row and column
9      for (int i = 0; i <= L1; i++) {
10         dp[i][0] = i;
11     }
12     for (int j = 0; j <= L2; j++) {
13         dp[0][j] = j;
14     }
15
16     // Calculate the minimum edit distance using dynamic
17     // programming
18     for (int i = 1; i <= L1; i++) {
19         for (int j = 1; j <= L2; j++) {
20             int cost = (S1[i - 1] == S2[j - 1]) ? 0 : 1;
21             dp[i][j] = min(min(dp[i - 1][j] + 1, dp[i][j - 1] + 1),
22                             dp[i - 1][j - 1] + cost);
23
24             // Check for character transposition
25             if (i > 1 && j > 1 && S1[i - 1] == S2[j - 2] && S1[i -
26                 2] == S2[j - 1]) {
27                 dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost);
28             }
29         }
30     }
31
32     // Return the minimum edit distance
33     return dp[L1][L2];
34 }
```

Листинг 2 – Итеративный алгоритм нахождения расстояния Дамерау-Левенштейна

В листинге 3 приведена реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

```
1  int DistanceSolver::recursiveDamerauLevenshtein(int L1, int L2) {
2      // Base case: If both strings are empty, return 0 (edit
        distance)
3      if (L1 == 0 && L2 == 0) return 0;
4
5      // If one of the strings is empty, return the length of the
        other string
6      if (L1 == 0) return L2;
7      if (L2 == 0) return L1;
8
9      // Calculate the cost for the current character pair
10     int cost = (S1[L1 - 1] == S2[L2 - 1]) ? 0 : 1;
11
12     // Calculate the edit distances for insertion, deletion, and
        substitution
13     int insertion = recursiveDamerauLevenshtein(L1, L2 - 1) + 1;
14     int deletion = recursiveDamerauLevenshtein(L1 - 1, L2) + 1;
15     int substitution = recursiveDamerauLevenshtein(L1 - 1, L2 - 1)
        + cost;
16
17     // Find the minimum edit distance among the three operations
18     int result = min(min(insertion, deletion), substitution);
19
20     // Check for character transposition
21     if (L1 > 1 && L2 > 1 && S1[L1 - 1] == S2[L2 - 2] && S1[L1 - 2]
        == S2[L2 - 1]) {
22         int transposition = recursiveDamerauLevenshtein(L1 - 2, L2
            - 2) + cost;
23         result = min(result, transposition);
24     }
25
26     return result;
27 }
```

Листинг 3 – Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна



В листинге 4 приведена реализация рекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна с кэшем.

```
1  int DistanceSolver::recursiveCacheDamerauLevenshtein(int L1, int L2
   ) {
2      if (cache[L1][L2] != INT_MAX) {
3          return cache[L1][L2];
4      }
5      if (L1 == 0) {
6          cache[L1][L2] = L2;
7          return L2;
8      }
9      if (L2 == 0) {
10         cache[L1][L2] = L1;
11         return L1;
12     }
13
14     int cost = (S1[L1 - 1] == S2[L2 - 1]) ? 0 : 1;
15     int insertion = recursiveCacheDamerauLevenshtein(L1, L2 - 1) +
16         1;
17     int deletion = recursiveCacheDamerauLevenshtein(L1 - 1, L2) +
18         1;
19     int substitution = recursiveCacheDamerauLevenshtein(L1 - 1, L2
20         - 1) + cost;
21     int result = min(min(insertion, deletion), substitution);
22
23     if (L1 > 1 && L2 > 1 && S1[L1 - 1] == S2[L2 - 2] && S1[L1 - 2]
24         == S2[L2 - 1]) {
25         int transposition = recursiveCacheDamerauLevenshtein(L1 -
26             2, L2 - 2) + cost;
27         result = min(result, transposition);
28     }
29
30     cache[L1][L2] = result;
31     return result;
32 }
```

Листинг 4 – Рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна с кэшем

## 4 Исследовательская часть

### 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся замерный эксперимент:

- операционная система Windows 11;
- память 16 ГБ;
- процессор 3,6 ГГц 6-ядерный процессор AMD Ryzen 5000 series 5.

Замеры проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только окружением и непосредственно выполняемой программой.

### 4.2 Пример работы программы

На рисунке 5 представлен пример работы программы. Вводится две строки, после чего выводится результат и время работы каждого алгоритма в микросекундах.

```
Введите первую строку S1: abacaba
Введите вторую строку S2: ababaga
Расстояние Левенштейна: 2
Расстояние Дameraу-Левенштейна: 2
Левенштейн: 4.746000e+03 тактов
Дameraу-Левенштейн: 7.770000e+03 тактов
Рекурсивный Дameraу-Левенштейн: 8.531670e+05 тактов
Рекурсивный Дameraу-Левенштейн с кэшем: 3.234000e+03 тактов
```

Рисунок 5 – Пример работы программы

### 4.3 Время выполнения реализованных алгоритмов

Введём следующие обозначения:

**Л** - Левенштейна

**ДЛ** - Дамерау-Левенштейн

**РДЛ** - Рекурсивный Дамерау-Левенштейн

**РДЛК** - Рекурсивный Дамерау-Левенштейн с кэшем

Таблица 1 – Результаты замеров времени реализованных алгоритмов в тактах процессора

Длина строк	Л	ДЛ	РДЛ	РДЛК
1	$1.720 \times 10^3$	$2.014 \times 10^3$	$87.255 \times 10^0$	$114.828 \times 10^0$
2	$1.483 \times 10^3$	$1.759 \times 10^3$	$23.594 \times 10^0$	$26.462 \times 10^0$
3	$2.797 \times 10^3$	$2.803 \times 10^3$	$146.202 \times 10^0$	$62.051 \times 10^0$
4	$3.084 \times 10^3$	$2.897 \times 10^3$	$719.301 \times 10^0$	$103.013 \times 10^0$
5	$4.516 \times 10^3$	$4.042 \times 10^3$	$3.699 \times 10^3$	$2.098 \times 10^3$
6	$3.984 \times 10^3$	$3.984 \times 10^3$	$17.288 \times 10^3$	$20.112 \times 10^3$
7	$3.785 \times 10^3$	$3.513 \times 10^3$	$83.077 \times 10^3$	$24.901 \times 10^3$
8	$6.053 \times 10^3$	$5.256 \times 10^3$	$553.444 \times 10^3$	$6.769 \times 10^3$
9	$8.088 \times 10^3$	$5.973 \times 10^3$	$2.640 \times 10^4$	$8.404 \times 10^3$
10	$8.570 \times 10^3$	$6.942 \times 10^3$	$1.433 \times 10^5$	$11.075 \times 10^4$

Замеры времени работы реализованных алгоритмов для определенной длины строк проводились 100 раз, при этом каждый раз строки генерировались случайно. В качестве результата, представленного в таблице 1, взято среднее время на каждой длине слова.

#### 4.4 Занимаемая память реализованных алгоритмов

В данном разделе представлены результаты замеров затрат памяти для различных алгоритмов на обработку строк разной длины. Таблица 2 демонстрирует объем памяти в байтах, потребляемый разными реализациями алгоритмов в зависимости от длины обрабатываемых строк.

Таблица 2 – Затраты памяти в байтах для различных алгоритмов в зависимости от длины строк

N	Л	ДЛ	РДЛ	РДЛК
1	28	28	32	64
2	48	48	32	140
3	76	76	32	224
4	112	112	32	352
5	156	156	32	544
6	208	208	32	800
7	268	268	32	1088
8	336	336	32	1376
9	412	412	32	1696
10	496	496	32	2032

#### 4.5 Сравнительный анализ алгоритмов

Рекурсивная реализация алгоритма Дамерау-Левенштейна (РДЛ) демонстрирует значительно худшую производительность по времени выполнения по сравнению с итеративными алгоритмами. Рекурсивные алгоритмы следует использовать только для обработки строк крайне малой длины (1-2 символа) или в ситуациях, когда доступно ограниченное количество оперативной памяти.

Алгоритм Дамерау-Левенштейна (ДЛ) и итеративная реализация с кэшированием (РДЛК) подходят для обработки строк разной длины. Однако следует обратить внимание, что ДЛ затрачивает больше памяти по сравнению с РДЛК, особенно при работе с длинными строками. РДЛК является предпочтительным выбором, когда требуется учитывать ошибки, связанные с транспозицией символов, и при этом ограничены объемы доступной памяти. Несмотря на увеличение затрат памяти при использовании итеративных алгоритмов, они предоставляют хорошую скорость выполнения, особенно при обработке строк средних и больших размеров.

В случаях, когда расстояние Дамерау-Левенштейна играет важную роль из-за ошибок, связанных с транспозицией символов, алгоритм Дамерау-Левенштейна (ДЛ и РДЛК) является наиболее предпочтительным выбором, несмотря на некоторое увеличение временных и памятных затрат.

#### **4.6 Вывод**

В результате анализа замеров времени выполнения и затрат памяти на различных алгоритмах были сделаны следующие выводы:

- Для обработки строк малой длины и при ограниченных объемах памяти рекомендуется использовать рекурсивные алгоритмы.
- Для строк средних и больших размеров и при наличии достаточной памяти, итеративные алгоритмы предоставляют лучшую производительность.
- При необходимости учета транспозиций символов и ограниченных ресурсах памяти алгоритм Дамерау-Левенштейна с кэшированием (РДЛК) является предпочтительным выбором.

В конечном итоге, выбор подходящего алгоритма зависит от требований вашей задачи, доступных ресурсов памяти и желаемой скорости выполнения.

## ЗАКЛЮЧЕНИЕ

Цель работы достигнута: изучены алгоритмы нахождения расстояния Левенштейна и Дамерау–Левенштейна.

В ходе выполнения лабораторной работы были решены все задачи:

- изучены алгоритмы нахождения расстояния Левенштейна и Дамерау–Левенштейна;
- применены методы динамического программирования для реализации алгоритмов;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по лабораторной работе.

Эксперименты показали, что наиболее затратный по времени рекурсивный алгоритм поиска расстояния Дамерау–Левенштейна без кеша, а наименее затратны итеративные алгоритмы. Менее затратными по памяти являются реализации итеративных алгоритмов. Самым затратным по памяти является реализация рекурсивного алгоритма поиска расстояния Дамерау–Левенштейна с кешированием.